Parallelization of NAS Benchmarks for Shared Memory Multiprocessors

Abdul Waheed and Jerry Yan[†]

NAS Technical Report NAS-98-010 March '98

{waheed,yan}@nas.nasa.gov
NAS Parallel Tools Group
NASA Ames Research Center
Mail Stop T27A-2
Moffett Field, CA 94035-1000

Abstract

This paper presents our experiences of parallelizing the sequential implementation of NAS benchmarks using compiler directives on SGI Origin2000 distributed shared memory (DSM) system. Porting existing applications to new high performance parallel and distributed computing platforms is a challenging task. Ideally, a user develops a sequential version of the application, leaving the task of porting the code to parallelization tools and compilers. Due to the simplicity of programming shared-memory multiprocessors, compiler developers have provided various facilities to allow the users to exploit parallelism. Native compilers on SGI Origin2000 support multiprocessing directives to allow users to exploit loop-level parallelism in their programs. Additionally, supporting tools can accomplish this process automatically. We experimented with these compiler directives and supporting tools by parallelizing sequential implementation of NAS benchmarks. Results reported in this paper indicate that with minimal effort, the performance gain is comparable with the hand-parallelized, carefully optimized, message-passing implementations of the same benchmarks.

† MRJ Technology Solutions, Inc., NASA Contract NAS2-14303, Moffett Field, CA 94035-1000

1 Introduction

Distributed Shared Memory (DSM) systems are becoming increasingly popular in high performance computing. Such systems can be considered scalable alternatives of conventional *Symmetric Multiprocessors* (SMPs) due to distributed memory. Additionally, DSM systems offer the ease of programming due to a global address spaces, similar to SMPs. There are at least three programming paradigms that can exploit parallelism offered by a DSM system: (1) explicit message-passing; (2) data-parallelism; and (3) compiler-directed multiprocessing. Explicit message-passing allows the users to write parallel programs with greater control over communication. This technique often involves domain decomposition, such that each processor in the system works on a part of the entire program data in a *Single Program, Multiple Data* (SPMD) paradigm. Intermediate results and synchronizations are accomplished through commonly used message-passing libraries, such as MPI [8]. Data-parallel programming languages allow the users to write SPMD programs without worrying about communication, which is handled by the compiler and its runtime system. The main source of parallelism is the program data, which can be distributed among different processors in a variety of ways. Data distribution is controlled through compiler directives. *High Performance Fortran* (HPF [5]) is a standard for these directives that have been used by several compiler developers.

Both message-passing and data-parallelism force a user to develop a parallel algorithm, which is a complex and challenging task. Ideally, a user would like to develop sequential code for a given application, leaving the task of porting the code to parallelization tools and compilers. Due to the simplicity of programming DSM systems, compiler developers have been investigating different techniques to exploit parallelism directly for such systems. This process can be accomplished automatically with a compiler or through some hints provided by the user to the compiler [11]. Some examples of automatic parallelization tools that transform sequential code to parallel code by inserting parallelization directives supported by the native compilers include: SUIF [2], Polaris [12], and KAP [7].

In this paper, we present our experiences of using native parallelization and optimization tools on SGI Origin2000. Origin 2000 is a DSM system with a *cache-coherent Non Uniform Memory Access* (ccNUMA) architecture. Each node of the system typically consists of two R10000 processors with two levels of separate data and instruction caches for each processor; and 512 MB of main memory shared between two processors on the node. Multiple system nodes are connected in a hypercube topology through a high speed network. Native tools that are of interest to our parallelization effort include: *Power Fortran Accelerator* (PFA), which can automatically insert parallelization directives in sequential code and transform the loops to enhance their performance; *Parallel Analyzer View* (PAV), which can annotate the results of dependence analysis of PFA and present them graphically; and *MipsPro Fortran77* compiler with MP runtime library to compile and execute the parallelized code. In addition to using these tools, we inserted some directive by hand to assist the compiler and tune the performance. We experimented with these compiler directives and supporting tools by parallelizing the sequential implementation of NAS benchmarks [4]. Our results indicate that with minimum effort, the performance is almost as good as the hand-parallelized, carefully optimized, message-passing version of the same benchmarks.

In section 2, we outline the directives-based parallelization paradigm. Section 3 provides details of our parallelization of NAS benchmarks for Origin2000. We compare the performance of parallelized code with hand-parallelized code in Section 4. Section 5 overviews related research efforts. We discuss our conclusions in Section 6.

2 Directives-Based Parallelization Methodology

A sequential program is first analyzed to discover: (1) loops that are the main source of parallelism; and (2) any dependencies among different loop iterations that inhibit parallelization of that loop for the sake of correctness. Based on this analysis, it may be possible to modify the code to remove dependencies. Parallelism is expressed simply by inserting appropriate compiler directives before a loop [9,11]. As Figure 1 indicates, this is essentially an iterative process of modifying the loop nests in the sequential code until most of the computationally expensive loops are parallelized. Finally, the parallelized code (i.e., sequential code with compiler directives) is compiled and linked with appropriate runtime libraries to execute on the target system.

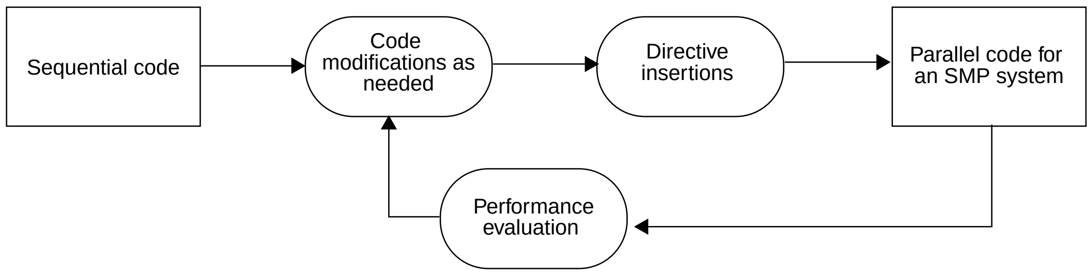


Figure 1. General methodology of parallelizing sequential code for shared-memory multiprocessors using compiler directives.

Directives-based parallelism is supported by the MP runtime library on Origin2000, which implements a fork-and-join paradigm of parallelism. A *master thread* initiates the program, creates multiple *slave threads*, schedules the iterations of parallelized loops on all the threads including itself, waits for the completion of a parallel loop by all the slave threads, and executes sequential portions of the program. Slave threads must wait for work (i.e., for parts of subsequent parallel loops) while the master thread is executing a sequential portion of the code.

3 Parallelization of Sequential NAS Benchmarks

NAS benchmarks consist of several Computational Fluid Dynamics (CFD) kernels and applications, frequently used to solve systems of partial differential equations that model the dynamics of a physical system. We select five of these benchmarks: BT, SP, CG, MG, and FT. These represent important classes of solvers for partial differential equations used in real CFD applications. Since, these solvers represent interesting compute-intensive parts of CFD applications, we selected them for this parallelization study.

3.1 Automatic Parallelization

The SP, CG, and MG benchmarks were parallelized using PFA and PAV. This resulted in parallelization of most of the loops that consumed significant portion of the entire execution. Flow diagrams in Figure 2 represent the key subroutines of benchmarks SP, CG, and MG that were parallelized.

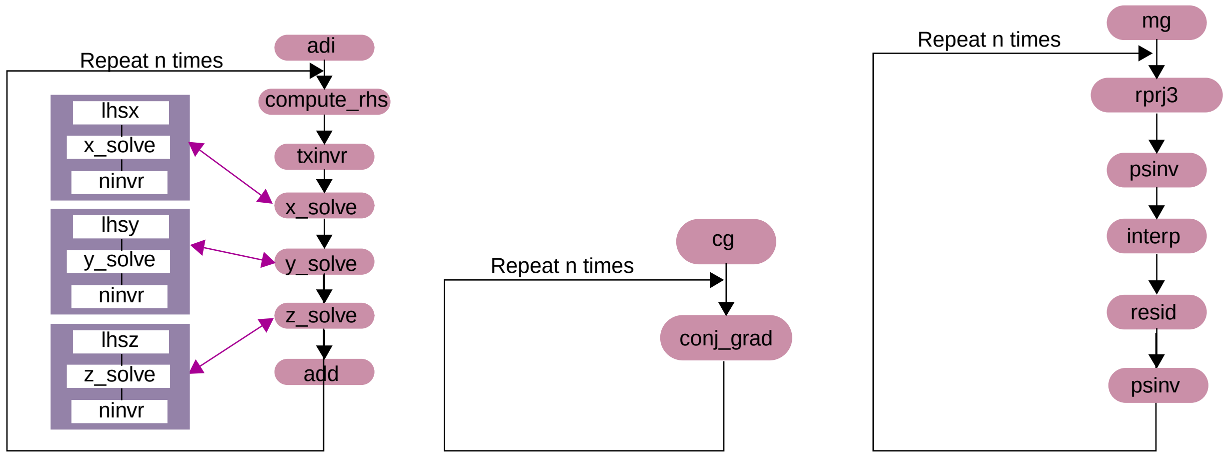


Figure 2. Code structure of sequential NAS benchmarks: (a) SP; (b) CG; and (c) MG.

SP is an application benchmark. The SP code solves a *Scalar Pentadiagonal* system of equations resulting from approximately factored, implicit, finite-difference discretization of the Navier-Stokes equations in three dimensions. The solution is based on an *Alternating Direction Implicit* (ADI) algorithm. This algorithm solves three sets of uncoupled systems of equations in *x*, *y*, and *z* directions. The main subroutines of SP, as illustrated in Figure 2(a), contain one or more loops whose iterations could be distributed among multiple processors.

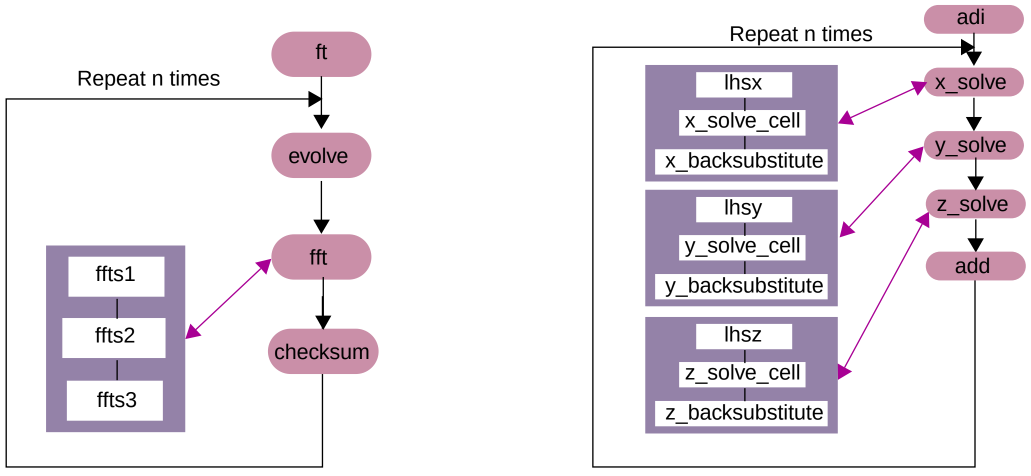
CG is a kernel benchmark based on a *Conjugate Gradient* method to compute an approximation to the smallest eigen value of a large, sparse, symmetric positive definite matrix. This kernel implements unstructured grid computations and communications. The CG code (see Figure 2(b)) consists of only one major loop that was identified and parallelized automatically by PFA.

MG uses a *Multi-Grid* method to compute the solution of the three-dimensional scalar Poisson equation. Four critical subroutines that use multi-grid operators on a grid perform the V-cycle algorithm (see Figure 2(c)): the smoother (*psinv*); the residual calculation (*resid*); the residual projection (*rprj3*); and the trilinear interpolation of the correction (*interp*). All four subroutines exhibit fine-grained loop-level parallelism that is detected by PFA to parallelize this code.

3.2 Hand-Coding of Parallelization Directives

PFA cannot automatically parallelize any significant number of time-consuming loops for FT and BT benchmarks due to two reasons: (1) source code shows complex dependences among iterations of a loop that require programmer input to resolve; and (2) a potentially parallel loop contains a procedure call, which may or may not have dependences on subsequent iterations of the loop.

PAV explained why a specific loop was not parallelized by PFA. For instance, BT is similar to SP in structure (see Figure 3(b)) but there are several procedure calls embedded in parallelizable loops of BT that PFA could not handle. In addition, both benchmarks required manual transformation of several loop nests to be parallelized. Data distribution directives were also added to improve the co-location of computation and data for these two benchmarks. After inserting parallelization directives by hand, this parallelized code was passed through PFA to obtain further loop optimization. Flow diagrams in Figure 3 represent the key subroutines of benchmarks FT and BT.



(a) **(b)**

Figure 3. Code structure of sequential NAS benchmarks: (a) FT and (b) BT.

FT is the computational kernel of a three dimensional *Fast Fourier Transform* (FFT)-based spectral method. The code computes FFT in the first, second, and third dimensions by calling subroutines ffts1, ffts2, and ffts3, respectively (see Figure 3(a)).

BT is an application benchmark. The code is similar to the SP code (see Figure 3(b)). It solves a *Block Tridiagonal* system of equations resulting from approximately factored, implicit, finite-difference discretization of the Navier-Stokes equations in three dimensions. The solution is based on an ADI algorithm that solves three sets of uncoupled, block tridiagonal systems of equations in *x*, *y*, and *z* directions. The main subroutines of BT contain loops with calls to three main parts of solver in each direction: formation of left-hand side (*lhz*); forward elimination for one block (*solve_cell*), and backsubstitution (*backsubstitute*). These subroutines offer sufficient parallelism that was exploited by inserting parallelization directives for key loops. A BLOCK data distribution in the *z*-direction was also added to improve the data locality.

In order to manually parallelize FT and BT, the following steps were taken: (1) inter-procedural analyses; (2) loop nest transformations; and (3) locality optimizations. Since we consider these steps to be generic and essential to parallelize any real application for a shared-memory multiprocessor, we present details of these steps in the rest of this subsection.

3.2.1 Inter-Procedural Analysis

Applications written in a structured manner often contain subroutine calls within some loops that are potentially parallelizable. It is safer for an automatic parallelization tool to assume that the subroutine calls embedded in a loop are not independent to avoid incorrect behavior. Several tools provide inter-procedural analysis, however, it is overly time-consuming for even modestly large codes. PFA does not provide inter-procedural analysis support. Unfortunately, it is not possible to leave the loops with subroutine calls unparallelized because they may represent a significant portion of the entire execution time. In such cases, the user is responsible to perform the inter-procedural analysis to decide whether or not a loop containing subroutine calls should be parallelized.

Figure 4(a) presents a typical block of code taken from FT. It consists of a loop nest, which is parallelizable except for a subroutine call. Even though the outer loop does not have any dependencies, it is not possible to determine whether the subroutine calls are independent from one iteration to another without inter-procedural analysis. This is a typical situation with numerous code blocks of FT and BT. In those cases, we analyze dependencies of the subroutine on subsequent iterations of the outer loop in the called function. If there are no dependencies, we manually parallelize the outer loop, as shown in Figure 4(b). Note that `C$DOACROSS` is the SGI Fortran77 loop parallelization directive [9].

```
do k = 1, d(3)
      do jj = 0, d(2) - fftblock, fftblock
         do j = 1, fftblock
            do i = 1, d(1)
               y(j,i,1) = x(i,j+jj,k)
            enddo
         enddo
         call cfftz (is, logd(1),
  >                   d(1), y, y(1,1,2))
         do j = 1, fftblock
            do i = 1, d(1)
               xout(i,j+jj,k) = y(j,i,1)
            enddo
         enddo
      enddo
   enddo
```

(a)

Inter-procedural analysis →

```
c$doacross local(k,jj,j,i,y)
      do k = 1, d(3)
         do jj = 0, d(2) - fftblock, fftblock
            do j = 1, fftblock
               do i = 1, d(1)
                  y(j,i,1) = x(i,j+jj,k)
               enddo
            enddo
            call cfftz (is, logd(1),
  >                      d(1), y, y(1,1,2))
            do j = 1, fftblock
               do i = 1, d(1)
                  xout(i,j+jj,k) = y(j,i,1)
               enddo
            enddo
         enddo
      enddo
```

(b)

Figure 4. An example of inter-procedural analysis during the parallelization of FT.

3.2.2 Loop Nest Optimization

It is customary to parallelize the outer-most loop in a loop nest to distribute substantial amount of computation to multiple processors. Indices in a loop nest are usually chosen to optimize the locality of data accessed from the loop. In many cases, some data accesses may have dependences due to the outer-most loop index while there are no dependencies due to at least one other loop index. For such cases, it may be possible to transform the loop nest such that the index with no dependences becomes the outer-most index to allow efficient parallelization of the entire loop nest. The user has to make a trade-off between the performance gain due to parallelization of the loop nest and the performance degradation due to non-optimal data locality. PFA and many other parallelization tools leave such decisions to the user.

Figure 5(a) presents a block of code taken from BT, which of a loop nest with one dependence due to the outermost index `k`. For a given value of `k`, an access to the array element `rhs(n,i,j,k+1)` may require a non-local memory access. If we parallelize the outermost loop, then accesses to the array `rhs` will need to be synchronized with other processors to ensure correctness. Considering that Fortran stores arrays in a column-major fashion, the loop indices are in an order that optimizes data locality. However, note that there is no dependences due to any other loop indices. Therefore, interchanging `j` and `k` indices will result in a minimum penalty of non-optimal data locality compared to any other permutation of loop indices. Additionally, we can now parallelize the outer-most loop and the performance gain due to more computation within the outer loop offsets the cost of non-optimal data locality (see Figure 5(b)).

```
do k=1,grid_points(3)-2
   do j=1,grid_points(2)-2
       do i=1,grid_points(1)-2
         do m=1,BLOCK_SIZE
           do n=1,BLOCK_SIZE
             rhs(m,i,j,k) = rhs(m,i,j,k)
  >                 - lhs(m,n,cc,i,j,k)*rhs(n,i,j,k+1)
           enddo
         enddo
       enddo
     enddo
   enddo
```

(a)

```
c$doacross local(j,k,i,m,n)
    do j=1,grid_points(2)-2
      do k=grid_points(3)-2
        do i=1,grid_points(1)-2
          do m=1,BLOCK_SIZE
            do n=1,BLOCK_SIZE
              rhs(m,i,j,k) = rhs(m,i,j,k)
   >                  - lhs(m,n,cc,i,j,k)*rhs(n,i,j,k+1)
            enddo
          enddo
        enddo
      enddo
    enddo
```

(b)

Figure 5. An example of loop nest optimization during parallelization of BT.

3.2.3 Data Locality Optimization

An important performance consideration for a DSM system is to locate data close to the computation to obtain reasonable performance. Non-local memory references and false-sharing are the main data locality bottlenecks that affect most of the shared-memory parallel programs. SGI Fortran77 compiler supports data `distribution` and data `affinity` directives to improve the locality of data close to the computation [9]. This data distribution is different from that supported by data-parallel languages. Data-parallel languages support the distribution of individual elements of arrays on to different nodes. However, the data distribution directives here support the distribution at a coarse granularity of pages of memory.

Figure 6(a) shows an example code taken from BT where data `distribution` and `affinity` directives were used to co-locate data with computation. The `BLOCK` distribution was used along one dimension of array `rhs` with the `affinity` clause (see Figure 6(b)) to ensure the co-location of specific pages of data with the computation. This significantly improved its performance (see Section 4).

```
c$doacross local(k,j,i,m,n)
do k=1,grid_points(3)-2
      do j=1,grid_points(2)-2
        do i=grid_points(1)-2,0,-1
          do m=1,BLOCK_SIZE
            do n=1,BLOCK_SIZE
              rhs(m,i,j,k) = rhs(m,i,j,k)
   >                  - lhs(m,n,cc,i,j,k)*rhs(n,i+1,j,k)
            enddo
          enddo
        enddo
      enddo
    enddo
```

(a)

```
c$distribute rhs(*,*,*,BLOCK)
c$doacross local(k,j,i,m,n)
c$&,       affinity(k) = data(rhs(m,i,j,k))
    do k=1,grid_points(3)-2
      do j=1,grid_points(2)-2
        do i=grid_points(1)-2,0,-1
          do m=1,BLOCK_SIZE
            do n=1,BLOCK_SIZE
              rhs(m,i,j,k) = rhs(m,i,j,k)
   >                  - lhs(m,n,cc,i,j,k)*rhs(n,i+1,j,k)
            enddo
          enddo
        enddo
      enddo
    enddo
```

(b)

Figure 6. Co-location of data with computation for parallelizing BT.

4 Evaluation of Parallelized Code

We evaluate the parallelized code from two perspectives: performance and level of effort. We quantitatively analyze the performance of directives-based parallelized programs by comparing their execution times with the hand-parallelized and optimized implementations of the same programs. In addition, we qualitatively evaluate the level of effort to parallelize sequential code using shared-memory multiprocessing directives.

4.1 Comparative Performance

NAS benchmarks were originally written as a suite of paper-and-pencil benchmarks to allow high-performance computing system vendors and researchers to develop their own implementations to evaluate specific architectures of their interest [4]. NAS also provides its own of hand-parallelized message-passing implementation of the benchmarks based on MPI message-passing library [10]. This implementation is carefully written and optimized for a majority of existing high performance computing platforms. Therefore, we compare the performance of our directive-based implementation of NAS benchmarks with the MPI-based hand-parallelized implementation for Origin2000. Class A benchmarks were used for comparisons reported in this subsection.

Figure 7 presents the comparison between automatically parallelized implementations of SP, CG, and MG with the hand-parallelized implementations of the same. For CG and MG, the performance of two implementations is comparable. However, the performance of directives-based parallelized SP degrades as the number of processors increases beyond eight while execution time for the MPI-based implementation continues to reduce with the number of processors. This difference is primarily due to superior data placement of the hand-parallelized SP resulting from a finer granularity of data distribution. As the number of processors increases, the amount of data owned by a processor reduces proportionately. This results in a better memory system performance. For the directives-based implementation, data is distributed at the granularity of pages. Therefore, as the number of processors increase, multiple processors have to access data from pages that they do not own locally, which adversely impact the overall execution time. In case of CG and MG, data locality does not become a bottleneck due to comparatively smaller size of code with smaller number of memory accesses. Therefore, performance remains comparable with the hand-parallelized implementations of CG and MG.

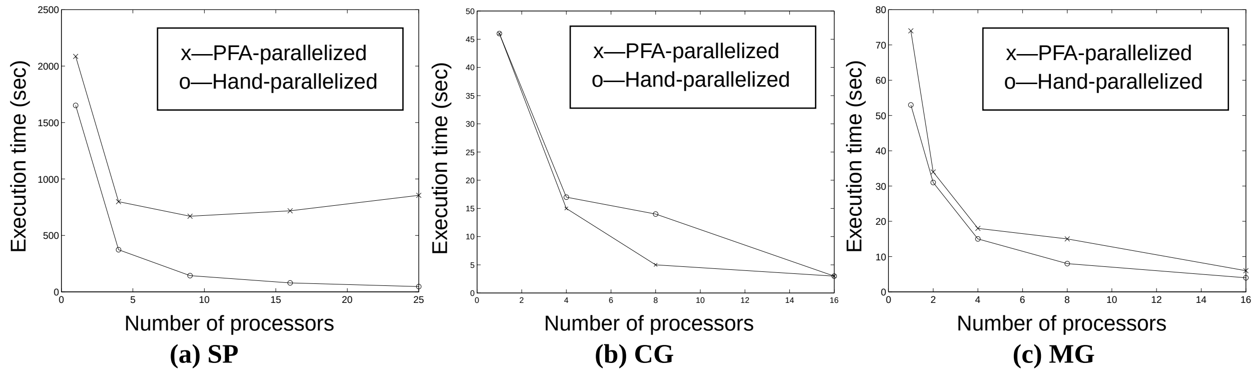
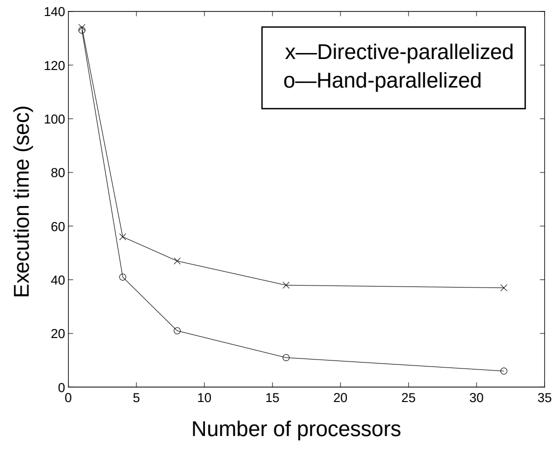
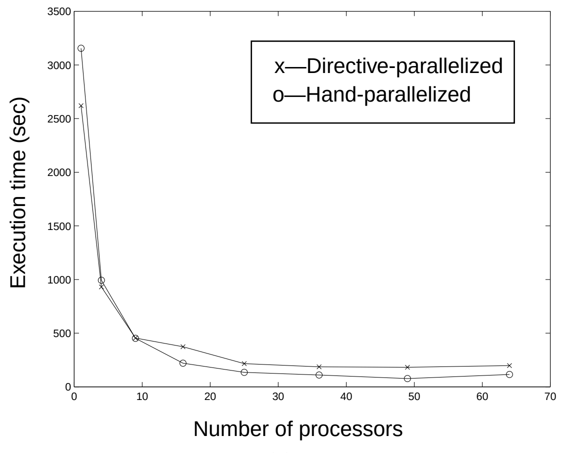


Figure 7. Performance comparison of automatic shared-memory multiprocessing directives-based parallelization of SP, CG, and MG benchmarks with MPI-based, hand-parallelized and -optimized current versions of the same benchmarks. These results are based on Class A benchmarks.

Figure 8 presents the comparison between directives-based parallelized versions of FT and BT with the hand-parallelized, MPI-based versions of the same. In both the cases, the performance improves with the number of processors. In case of FT, the MPI-based implementation proves to be superior to the shared-memory implementation due to data placement. In case of BT, the data locality was meticulously tuned for almost all the parallelized loops to ensure that each loop iteration is scheduled at a processor that owns elements of an array accessed during those iterations. Consequently, the performance of BT is comparable to its hand-parallelized implementation.



(a) FT



(c) BT

Figure 8. Performance comparison of shared-memory multiprocessing directives-based parallelization of FT and BT benchmarks with MPI-based, hand-parallelized and -optimized current versions of the same benchmarks. These results are based on Class A benchmarks.

4.2 Level of Effort

Parallelization of a sequential application is a non-trivial task that often requires hundreds of man-hours of effort [13]. For instance, hand-parallelization of NAS benchmark suite took more than ten man-years. However, it is hard to separate time spent on parallelization and time spent on tuning. It takes only a matter of minutes to automatically parallelize SP, CG, and MG using native tools on Origin2000. For FT and BT, about two weeks were spent to analyze and manually parallelize them using shared-memory multiprocessing directives. For BT, an additional week was required to tune its performance and scalability by inserting appropriate data placement directives. It should be noted that this included “learning time” as we were not intimately familiar with the application-domain details of these programs during their directives-based parallelization. Based on our qualitative assessment, this level of effort is considerably less than it would have been necessary to parallelize these benchmarks using explicit message-passing.

5 Related Work

Several research efforts have focused on parallelizing sequential programs for shared-memory multiprocessors. These efforts are becoming increasingly important due to the revival of shared-memory multiprocessors with improved scalability via distributed memory and hardware cache-coherence. SUIF compiler system incorporates various modules that can be used to analyze the sequential program, parallelize the loops, distribute program arrays, and perform inter-procedural analysis [2,3]. Polaris is another parallelizing compiler that can generate parallelized code for SMPs [12,14]. CAPTools is a semi-automatic parallelization tool that transforms a sequential program to a message-passing program by user-directed distribution of arrays [6]. Fortran-D [1] and various implementations of High Performance Fortran

(HPF [5]) are examples of parallelizing compilers that work for sequential programs that can benefit from data parallelism. KAP [7] and PFA [9] are examples of commercial parallelization tools for SMPs. Our experiences with most of these tools to parallelize sequential NAS benchmarks will be reported elsewhere. Based on this experience and results reported in this paper, we consider that tools for SMPs are simple to learn and use and their performance is promising.

6 Discussion and Conclusions

This study focused on parallelization of sequential NAS benchmarks using compiler directives for shared memory multiprocessing on Origin2000. Our experiences indicate that directives based parallelization is simple and requires minimal effort on the part of users by exploiting shared address space of the underlying architecture. Performance can be improved incrementally by enhancing the efficient use of memory hierarchies, especially caches.

Parallelization of sequential code for shared-memory systems has been a thoroughly researched area. Parallelization tools are starting to become popular due to the revival of shared-memory architecture by DSM systems. Use of directives-based parallelism has been limited due to portability issues. Almost every vendor of a shared-memory system offers its own extension of Fortran77 language via parallelization directives. These directives are not portable from one shared-memory system to another. Recently, several vendors have initiated a standardization efforts for shared-memory parallelization directives in the form of OpenMP standard [11]. OpenMP has proposed an API for compiler directives that can be used across shared-memory platforms. Introduction of such standards is a promising development that may simplify the portability issues.

An added advantage of directives-based parallelism is that the program can be compiled as a sequential program because directives appear as comments to the compiler without appropriate flag. Therefore, there is no need to maintain a separate sequential version of an application. Additionally, programs are not required to be recompiled for executing on a different number of processors. The runtime system determines dynamically determines the number of processors and appropriately schedules the parallelized loop iterations.

Directives-based parallelism is essentially a fine-grained parallelism that works at the level of individual loop iterations. This is greatly different from conventional coarse-grained parallelism at the level of processes or threads. When it is implemented carefully, it ensures much better load-balance compared to the conventional message-passing or data-parallel techniques. On the other hand, the user is required to spend additional time to ensure proper data locality to obtain performance that is comparable to hand-parallelized, message-passing based implementation.

References

[1] V. Adve, J-C. Wang, J. Mellor-Crummey, D. Reed, M. Anderson, and K. Kennedy, “An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs,” *Proceedings of Supercomputing ‘95*, San Diego, CA, December 1995.

[2] S. P. Amarasinghe, J. M. Anderson, M. S. Lam and C. W. Tseng, “The SUIF Compiler for Scalable Parallel Machines,” *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing*, July, 1995.

[3] Jennifer-Ann M. Anderson, “Automatic Computation and Data Decomposition for Multiprocessors,” Technical Report CSL-TR-97-719, Computer Systems Laboratory, Dept. of Electrical Eng. and Computer Sc., Stanford University, 1997.

[4] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow, “The NAS Parallel Benchmark 2.0,” Technical Report NAS-95-020, December 1995.

[5] High Performance Fortran Forum. High Performance Fortran Language Specification, Version 1.0. Scientific Programming, 2(1 & 2), 1993.

[6] C. S. Ierotheou, S. P. Johnson, M. Cross, and P. F. Leggett “Computer aided parallelisation tools (CAPTools)—conceptual overview and performance on the parallelisation of structured mesh codes” *Parallel Computing*, Vol.22, 1996, pp.163-195.

[7] Kuck & Associates, Inc., “Experiences With Visual KAP and KAP/Pro Toolset Under Windows NT,” Technical Report, Nov. 1997.

[8] Message Passing Interface Forum, “MPI: A Message-Passing Interface Standard,” May 5, 1994.

[9] *MIPSpro Fortran77 Programmer’s Guide*, Silicon Graphics, Inc. Available on-line from: http://techpubs.sgi.com/library/dynaweb_bin/0640/bin/nph-dynaweb.cgi/dynaweb/SGI_Developer/MproF77_PG/@Generic__BookView.

[10] NAS Parallel Benchmarks. Available on-line from: http://science.nas.nasa.gov/Software/NPB.

[11] *OpenMP: A Proposed Standard API for Shared Memory Programming*, Oct. 1997. Available on-line from http://www.openmp.org.

[12] David A. Padua, Rudolf Eigenmann, Jay Hoeflinger, Paul Petersen, Peng Tu, Stephen Weatherford, and Keith Faigin, “Polaris: A New-Generation Parallelizing Compiler for MPPs,” Technical Report CSRD # 1306, University of Illinois at Urbana-Champaign, June 15, 1993.

[13] Cherri M. Pancake, “The Emperor Has No Clothes: What HPC Users Need to Say and HPC Vendors Need to Hear,”, *Supercomputing ‘95*, invited talk, San Diego, Dec. 3–8, 1995.

[14] Insung Park, Michael J. Voss, and Rudolf Eigenmann, “Compiling for the New Generation of High-Performance SMPs,” Technical Report, Nov. 1996.

	NAS TECHNICAL REPORT
	Title: **Parallelization of NAS Benchmarks for Shared Memory Multiprocessors**
	Author(s): Abdul Waheed and Jerry Yan
Two reviewers must sign.	Reviewers: “I have carefully and thoroughly reviewed this technical report. I have worked with the author(s) to ensure clarity of presentation and technical accuracy. I take personal responsibility for the quality of this document.” Signed: ______________________________ Name: _H. Jin____________________ Signed: ______________________________ Name: _M. Hribar________________
After approval, assign NAS Report number.	Branch Chief: Approved: ____________________________
Date:	NAS ReportNumber: NAS-98-010